

# The Fletcher Checksum

Brian Mearns

June 12, 2012

## Abstract

This document provides just a little bit of information on the Fletcher family of checksum algorithms.

## 1 Algorithm Overview

The Fletcher checksum algorithm acts on a sequence of *words*, and produces as output a pair of words which are called the *sum* and *checksum* values (the latter term will be avoided in this paper to avoid confusion with the *check* values introduced in the next section).

We can define a *word* generically to be some numerical container which can unambiguously store any integer in the closed range  $[0, W]$ , for some integer  $W > 0$ . As the algorithm is commonly employed in digital communications, a word would typically be an 8-bit or a 16-bit value (with  $W$  equal to  $2^8 - 1$  and  $2^{16} - 1$ , respectively).

Once the word is defined, the Fletcher algorithm is further defined by a parameter  $M$ , which is called the *modulus value*. The final sum values which are produced by the algorithm will be in modulo- $M$  reduced closed range  $[0, M - 1]$ , thus  $M$  is virtually always chosen such that  $M \leq W + 1$  so that the sum values will each have a maximum value of  $W$  and will therefore each fit into a single word.

The Fletcher checksum algorithm computes two summations over the sequence of input words: a *simple sum* and a *compound sum*. The simple sum is just the summation of all the words in the sequence, and the compound sum is the sum of all the partial simple sums.

More formally, both sums are initially set to 0. For each word in the input sequence (in order), the value of the word is first added to the simple sum, and then the simple sum is added to the compound sum. In pseudo code:

```
s1 := 0  #simple sum
s2 := 0  #compound sum
for d in input_seq:
    s1 := s1 + d
    s2 := s2 + s1
done
```

The output sums are simply the simple sum and the compound sum, each reduced modulo  $M$ . Strictly speaking, the modulo-reduction does not need to be performed until the end, but that assumes that the variables  $s1$  and  $s2$  will not roll over during the calculation. The reduction can occur as often as desired, and if there is any risk of a roll over happening (as would normally be the case for a digital implementation), it must be performed before any roll over occurs. Typical implementations will simply do reduction at every step:

```
s1 := 0 #simple sum
s2 := 0 #compound sum
for d in input_seq:
    s1 := (s1 + d) % M
    s2 := (s2 + s1) % M
done
```

You could conceivably add logic to test for a roll over before performing the addition, and only reducing the existing sum values if necessary, but the overhead of branching is typically worse than the overhead of performing a modulus operation.

Note that roll over or truncation is often used in computer code as a substitute for modulus reduction, but this only works when the modulus value is a power of 2. In any other case, a rollover is *not* the same as a modulus reduction and will break the algorithm.

One important thing to note from this is that the summations must be performed using a type large enough to prevent roll over since the sum must be resolved before the reduction. For instance, if the word is an 8-bit type, the sum of two words may need as many as 9-bits to avoid roll over. This is typically accommodated simply by giving the sum variables,  $s1$  and  $s2$ , types that are large enough to hold the sum of two words.

## 1.1 Choice of Modulo Values

One purpose of the modulo reduction is to ensure that the resulting sums will each fit into a single word. However, in doing so, you are effectively throwing away a potentially large amount of information about the input stream. In the simplest sense, a sum which is not reduced (and does not overflow) can take on infinitely many values to represent the infinite number of possible input sequences (although the former infinity is still smaller than the latter since multiple input sequences can yield the same sum). Once the sum is reduced, it can only represent  $M$  different values, which aliases even more possible input sequences to the same checksum.

None the less, there are good ways to reduce the sum, and there are bad ways. No matter what, you are effectively cutting bits off the top of the sum. The difference between a good reduction and a bad reduction is what you do with these chopped off bits. A bad reduction throws them away, while a good reduction mixes them back into the remaining bits.

A bad reduction is done in a binary implementation by using a power of 2 as the modulus value,  $M$ . In this case, the modulus reduction simply cuts off all the upper bits and discards them.

A good reduction will use a non-power of 2 modulus value for  $M$ , such that the reduction will mix the chopped off upper bits back into the lower bits.

At the same time, the modulus value should be large enough to maximize the number of possible checksums that can be produced to limit aliasing. For binary implementations with a  $k$ -bit word, a modulus value of  $M = 2^k - 1$  is commonly used (i.e., the maximum value of a single word).

### 1.1.1 Choice Based on Ease of Implementation

Of course, some modulus values may also lead to easier implementation than others, sometimes at the cost of reduced error detection capabilities. In the most obvious case, when the modulus value is a power of 2, modulus reduction is performed by simply chopping off the upper bits. In a binary implementation, this can be done generically with a bit mask, or in the specific case when  $M = W + 1$ , the reduction occurs automatically by rollover.

However, in some cases, the commonly used modulus value of  $2^k - 1$  (for a  $k$ -bit word) can also led to easier implementations (though not as easy as a  $2^k$  modulus), without impairing the error detecting capabilities. Specifically,  $X \% (2^k - 1)$  is equal to  $X \% 2^k$  when  $X < 2^k - 1$ , and is equal to  $(X + 1) \% 2^k$  otherwise. Therefore, the modulus-reduction operation (which may involve a lengthy division instruction) can be effectively replaced by a right-shift and a second addition (or an addition-with-carry where available).

Note that this is not strictly a reduction modulu  $2^k - 1$ , but it is effectively the same; the difference is that a value equal to  $2^k - 1$  will not be reduced to 0, it will stay  $2^k - 1$ . While this still fits into a  $k$ -bit word, it is not strictly equal to the value prescribed by the Fletcher algorithm. To match the Fletcher algorithm, the final sums will need to be properly reduced.

```
uint8_t d
uint16_t s0 := 0
uint16_t s1 := 0
for d in input_seq:
    s0 := s0 + d
    s0 := s0 + (s0 >> 8)    #Add carry-bit back in

    s1 := s1 + d
    s1 := s1 + (s1 >> 8)    #Add carry-bit back in
done

#Final reduction to properly match Fletcher:
s0 := s0 % 255
s1 := s1 % 255
```

## 1.2 Word-size variants

Commonly seen instances of the Fletcher checksum algorithm are those referred to as “Fletcher-16” and “Fletcher-32”. Each are named according to the *total* number of bits in the produced checksum. Thus “Fletcher-16” operates on a sequence of 8-bit words and produces two 8-bit sum values with the implicit modulus value  $M = 255$ ; and “Fletcher-32” operates on a sequence of 16-bit words and produces two 16-bit sum values with the implicit modulus value  $M = 65535$ .

## 2 Zero-Sum Check Values

A variation of the algorithm is to generate a pairs of words which, when appended to the end of the input stream produce an augmented stream whose Fletcher checksum (using the same parameters) is 0. We refer to these two words as the *check values*, as distinct from the *sum values* produced by the standard Fletcher algorithm already described.

To understand how to generate these check values, we must simply consider the effect they will have on the checksum when appended to the end of the data stream, noting that we want the resulting sum values to both be 0.

Assuming that the original input stream produces sum values of  $s_1$  and  $s_2$  for the simple and compound sum, respectively, then we will generate a pair of check values  $c_1$  and  $c_2$  such that:

$$s_1 + c_1 + c_2 \equiv 0 \quad (1)$$

$$s_2 + (s_1 + c_1) + (s_1 + c_1 + c_2) \equiv 0 \quad (2)$$

where the equivalency is understood to be under the modulus value  $M$  used to compute the Fletcher sum values  $s_1$  and  $s_2$ .

Equation 1 shows that we want the simple sum of the augmented stream to equal 0. Remember the simple sum is just the sum, modulo  $M$ , of all the words in the stream, and  $s_1$  is the simple sum of the original data stream. The augmented stream is formed by appending our check words,  $c_1$  and  $c_2$  to the original, so the simple sum of the augmented stream is found by simply adding  $c_1$  and  $c_2$  to  $s_1$ .

Likewise, equation 2 is the compound sum of the augmented stream: we start with the compound sum of the original data stream,  $s_2$ , and to that we add the final two simple sums of the augmented stream, which are found by adding first  $c_1$  and then  $c_2$  to the exiting simple sum  $s_1$  (as described in the previous paragraph).

From equation 1, we know that the final simple sum,  $s_1 + c_1 + c_2$  is equivalent to 0 under modulus  $M$ , so we can remove those terms from equation 2:

$$s_2 + (s_1 + c_1) \equiv 0 \quad (3)$$

Because the equivalency is under modulu  $M$ , we can just as well write:

$$s_2 + (s_1 + c_1) \equiv M \quad (4)$$

From this, we can replace the equivalency with equality by reducing each side by modulo  $M$ . Of course if we reduce the right side, it will be

back to 0. Alternatively, we can leave the right side as is, and reduce the left selectively:

$$((s_2 + s_1) \% M) + c_1 = M \quad (5)$$

Looking at equation 5, we have reduced the sum  $s_2 + s_1$  by modulo  $M$ , and so this sum is no less than 0 and strictly less than  $M$ . For this equation to be valid based on equation 4, it is sufficient to limit the value of  $c_1$  to the closed range  $[1, M]$ , which we will see is workable in all cases.

We can therefore rewrite equation 5 to get a formula for the value of  $c_1$ :

$$c_1 = M - ((s_2 + s_1) \% M) \quad (6)$$

To find a formula for  $c_2$ , we go back to equation 1, and perform a similar set of reductions:

$$s_1 + c_1 + c_2 \equiv 0 \quad (7)$$

$$s_1 + c_1 + c_2 \equiv M \quad (8)$$

$$((s_1 + c_1) \% M) + c_2 = M \quad (9)$$

$$c_2 = M - ((s_1 + c_1) \% M) \quad (10)$$

Equation 7 is an exact copy of the original equation 1 for reference. In equation 8, we once again replaced an equivalency to 0 under modulo  $M$  with an equivalency to  $M$ , and in equation 9 we again replaced this equivalency with equality by selectively reducing the left side of the equation. As with  $c_1$ , this operation simply limits the value of  $c_2$  to the closed range  $[1, M]$ . The last line, equation 10, rewrites the equation to provide a formula for the value of  $c_2$ .

Summarizing, we have the following formula for calculating the check word values  $c_1$  and  $c_2$ :

$$c_1 := M - ((s_2 + s_1) \% M)$$

$$c_2 := M - ((s_1 + c_1) \% M)$$

Depending on the nature of the application, it may be necessary to reduce these values by modulo  $M$ , to fit them into the proper closed range  $[0, M - 1]$ . However, this should not generally be necessary. The modulo parameter  $M$  under which the Fletcher checksum is performed is generally defined to be no greater than  $W + 1$ , where  $W$  is the maximum value that can be stored in one word (this is so that the sum values will each fit into one word as well).

In the limiting case, when  $M = W + 1$ , the maximum value for the check values  $c_1$  and  $c_2$  will likewise be  $W + 1$ , which doesn't fit into a word. However, if you simply assign (or cast) these values into a word, they should be truncated to 0, which is the same as reducing it modulo  $M$ . In all other cases ( $M < W + 1$ ), the check values  $c_1$  and  $c_2$  will necessarily fit into a word, so no reduction is necessary (although they will not necessarily be strictly in the modulo- $M$  reduced range, note that, in general, the words of the original data stream will not be either).